Section Solutions 3

Problem One: Splitting the Bill

The insight that we used in our solution is that the first person has to pay some amount of money. We can't say for certain how much it will be, but we know that it's going to have to be some amount of money that's between zero and the full total. We can then try out every possible way of having them pay that amount of money, which always leaves the remaining people to split up the part of the bill that the first person hasn't paid.

```
/**
 * Lists off all ways that the set of people can pay a certain total, assuming
 * that some number of people have already committed to a given set of payments.
 *
 * @param total The total amount to pay.
 * @param people Who needs to pay.
 * @param payments The payments that have been set up so far.
 */
void listPossiblePaymentsRec(int total, const Set<string>& people,
                             const Map<string, int>& payments) {
    /* Base case: if there's one person left, they have to pay the whole bill. */
    if (people.size() == 1) {
        auto finalPayments = payments;
        finalPayments[people.first()] = total;
        cout << finalPayments << endl;
    }
    /* Recursive case: The first person has to pay some amount between 0 and the
     * total amount. Try all of those possibilities.
     */
    else {
        for (int payment = 0; payment <= total; payment++) {
            /* Create a new assignment of people to payments in which this first
             * person pays this amount.
             */
            auto updatedPayments = payments;
            updatedPayments[people.first()] = payment;
            listPossiblePaymentsRec(total - payment, people - people.first(),
                                    updatedPayments);
        }
    }
}

void listPossiblePayments(int total, const Set<string>& people) {
    /* Edge cases: we can't pay a negative total, and there must be at least one
     * person.
     */
    if (total < 0) error("The coffee shop just paid you to drink coffees?");
    if (people.isEmpty()) error("Dine and dash?");

    listPossiblePaymentsRec(total, people, {});
}
```

Problem Two: Ordering Prerequisites

There are a number of different ways to do this. The key insight we used in this solution is that at each point in time, we can choose any task to do as the next task provided that we've already handled all its prerequisites. That gives us some number of things to try at each step, so we'll try doing each of them first.

```
void listLegalOrderingsRec(const Map<string, Set<string>>& prereqs,
                           const Vector<string>& tasksPerformed,
                           const Set<string>& tasksRemaining) {
    /* Base case: If there are no remaining tasks, output this as one possible
     * option.
     */
    if (tasksRemaining .isEmpty()) {
        cout << classesTaken << endl;
        return;
    }

    /* Otherwise, try adding in as a possible next task all tasks that have had
     * all their prerequisites satisfied.
     */

    /* For convenience, convert the vector of tasks into a set. */
    Set<string> tasksDone;
    for (string task: tasksPerformed) {
        tasksDone += task;
    }

    for (string task: remainingTasks) {
        if (prereqs[task].isSubsetOf(tasksDone)) {
            auto newTasks = tasksPerformed; // See Q1 for auto
            newTasks += task;
            listLegalOrderingsRec(prereqs, newTasks, tasksRemaining - task);
        }
    }
}

void listLegalOrderingsOf(const Map<string, Set<string>>& prereqs) {
    String<string> tasks;
    for (string task : prereqs) {
        tasks.add(task);
    }
    listLegalOrderingsRec(prereqs, {}, tasks);
}
```

Problem Three: Change We Can Believe In

The idea behind this solution is the following: if we need to make change for zero cents, the only (and, therefore, best!) option is to use 0 coins. Otherwise, we need to give back at least one coin. What's the first coin we should hand back? We don't know which one it is, but we can say that it's got to be one of the coins from our options and that that coin can't be worth more than the total. So we'll try each of those options in turn, see which one ends up requiring the fewest coins for the remainder, then go with that choice. The code for this is really elegant and is shown here:

```
/**
 * Given a collection of denominations and an amount to give in change, returns
 * the minimum number of coins required to make change for it.
 *
 * @param cents How many cents we need to give back.
 * @param coins The set of coins we can use.
 * @return The minimum number of coins needed to make change.
 */
int fewestCoinsFor(int cents, const Set<int>& coins) {
    /* Base case: You need no coins to give change for no cents. */
    if (cents == 0) {
        return 0;
    }
    /* Recursive case: try each possible coin that doesn't exceed the total as
     * as our first coin.
     */
    else {
        int bestSoFar = cents + 1; // Can never need this many coins; see why?
        for (int coin: coins) {
           /* If this coin doesn't exceed the total, try using it. */
           if (coin <= cents) {
               bestSoFar = min(bestSoFar, fewestCoinsFor(cents - coin, coins));
           }
        }
        return bestSoFar + 1; // For the coin we just used.
    }
}
```

We asked whether memoization would be appropriate here, and the answer is "yes, definitely!" Imagine, for example, that we're using this algorithm on US coins, and we want to see the fewest number of coins required to make change for 10¢. Our options include first using a dime, first using a nickel, and first using a penny. Both of those latter two routes will eventually want to know the best way to make change for 5¢, the case where we use a nickel immediately needs to know this, and the case where we first use a penny will want to know how to do this for 9¢, which eventually needs to know 8¢, etc. down to 5¢. Without using memoization, we'd end up with a ton of redundant computation, which would slow things down dramatically. With memoization, this will be lightning fast for most numbers!

Here's what this might look like:

```
/**
 * Given a collection of denominations and an amount to give in change, returns
 * the minimum number of coins required to make change for it. This uses a table
 * to memoize its results.
 *
 * @param cents How many cents we need to give back.
 * @param coins The set of coins we can use.
 * @param memoizedResults A table mapping numbers of cents to the minimum number
 *                        of coins needed to make that total, for memoization.
 * @return The minimum number of coins needed to make change.
 */
int fewestCoinsForMemoized(int cents, const Set<int>& coins,
                           Map<int, int>& memoizedResults) {
    /* Base case: You need no coins to give change for no cents. */
    if (cents == 0) {
        return 0;
    }
    /* Base case: if we already know the answer, just return it! */
    else if (memoizedResults.containsKey(cents)) {
        return memoizedResults[cents];
    }
    /* Recursive case: try each possible coin that doesn't exceed the total as
     * as our first coin.
     */
    else {
        int bestSoFar = cents + 1; // More than we ever need; see why?
        for (int coin: coins) {
           /* If this coin doesn't exceed the total, try using it. */
           if (coin <= cents) {
               int needed = fewestCoinsForMemoized(cents - coin, coins,
                            memoizedResults);
               bestSoFar = min(bestSoFar, needed);
           }
        }

        /* Store the result for later. */
        int result =  bestSoFar + 1;
        memoizedResults[cents] = result;
        return result;
    }
}

/**
 * Given a collection of denominations and an amount to give in change, returns
 * the minimum number of coins required to make change for it.
 *
 * @param cents How many cents we need to give back.
 * @param coins The set of coins we can use.
 * @return The minimum number of coins needed to make change.
 */
int fewestCoinsFor(int cents, const Set<int>& coins) {
    Map<int, int> memoizedResults;
    return fewestCoinsForMemoized(cents, coins, memoizedResults);
}
```

Problem Four: Member of the Wedding

Here's one way to think about this problem. If everyone is already seated, you're done! Just score how good your seating arrangement is so far. If not, then pick someone who isn't seated and think about all the places you could seat her. She has to go somewhere, after all! Then take a look and see which of those choices ends up working out best overall and go with it! With that in mind, here's one possible solution:

```
/**
 * Given a list of guests, an index into that list, a partial seating chart,
 * and the capacity of each table, returns the best possible seating chart we
 * can make by placing the remaining people into seats.
 *
 * @param guests The guests at the wedding.
 * @param index The index of the next person to place.
 * @param seatsSoFar The partial seating chart.
 * @param tableCapacity How much seats there are at each table.
 * @return The optimal seating chart we can make by adding people in to the tables
 *         given the existing seating chart.
 */
Map<string, Vector<string>>
bestSeatingRec(const Vector<string>& guests, int index,
               const Map<string, Vector<string>>& seatsSoFar, int tableCapacity) {
    /* Base Case: If everyone's seated, whatever we have is the best option. */
    if (index == guests.size()) {
        return seatsSoFar;
    }
    /* Otherwise, figure out where we're going to put the next person. */
    else {
        Map<string, Vector<string>> bestSoFar;
        int bestScore = -1; // Sentinel; will be overwritten

        /* Try each table that still has space for this person. */
        for (string table: seatsSoFar) {
            if (seatsSoFar[table].size() < tableCapacity) {
                /* Put this person at this table. To do so, we'll make a copy of
                 * the seating chart and add this person to this table.
                 */
                auto newChart = seatsSoFar;
                newChart[table] += guests[index];

                /* See how well we can do. */
                auto thisChoice = bestSeatingRec(guests, index + 1, newChart,
                                                 tableCapacity);

                /* If this is the best we've found, remember that. */
                if (scoreFor(thisChoice) > bestScore) {
                    bestSoFar = thisChoice;
                    bestScore = scoreFor(thisChoice);
                }
            }
        }

        return bestSoFar;
    }
}
```

```
/**
 * Given a list of guests, tables, and table capacities, returns the best seating
 * arrangement for the wedding.
 *
 * @param guests The guests at the wedding.
 * @param tableNames The names of each of the tables.
 * @param tableCapacity How much seats there are at each table.
 * @return The optimal seating chart.
 */
Map<string, Vector<string>>
bestSeatingArrangementFor(const Vector<string>& guests,
                          const Vector<string>& tableNames,
                          int tableCapacity) {
    Map<string, Vector<string>> tables;
    for (string tableName: tableNames) {
        tables[tableName] = {}; // No one is initially sitting here
    }
    return bestSeatingRec(guests, 0, tables, tableCapacity);
}
```